Laserengraveanalytics_notebook

New notebook

Customer Data Transformation

- Read the customer_data csv source file from which Churn Analytics is implemented
- Data is validated
- Error Handling: Exception rows are branched into an 'error' table
- Business Rules are applied
- Data distribution profile for Age, Income
- Holiday flag is include
- Cleaned data is saved to [customer_data_cleaned] table in Lakehouse

In [2]:
```
# Welcome to your new notebook
# Type here in the cell editor to add code!

from pyspark.sql.functions import col, to_date, when, lit, regexp_replace, to_ti
from pyspark.sql import SparkSession
from datetime import datetime, timedelta
from pyspark.sql.types import IntegerType, DoubleType
import requests

# Load raw data
#csv_path = "Files/customer_data.csv"
#df_raw = spark.read.option("header", True).csv(csv_path)
```

StatementMeta(, 9f38f95a-4ff1-4a42-bcd1-c8f923dbf6bc, 3, Finished, Available, Fin
ished)

- **Start Treating Data, Intermediate staging****
- Implement Rules**
- No missing customer_id or email
- No duplicate rows
- Trim white spaces in column names

In [3]:
```
# Read CSV
csv_path = "Files/customer_data.csv"
df = spark.read.option("header", True).csv(csv_path)

# Drop rows with nulls in critical columns
df_clean = df.dropna(subset=["customer_id", "email"])

# Remove duplicates
df_clean = df_clean.dropDuplicates()

'''
Beware of name conflicts caused by specifically, the built-in Spark function col
Where I assigned a variable called 'col' col is a function in spark
'''

# Trim whitespace from string columns
```

```
from pyspark.sql.functions import trim
string_cols = [f.name for f in df_clean.schema.fields if f.dataType.simpleString
for c in string_cols:
    df_clean = df_clean.withColumn(c, trim(df_clean[c]))
```

StatementMeta(, 9f38f95a-4ff1-4a42-bcd1-c8f923dbf6bc, 4, Finished, Available, Fin
ished)

- **Clean Data, Intermediate staging
- Implement Rules**
- No missing customer_id or email
- No duplicate rows

In [4]:
```
# Read and clean CSV
#df = spark.read.option("header", True).csv("Files/customer_data.csv")   #alread
#df_clean = df.dropna(subset=["customer_id", "email"]).dropDuplicates()  #alread
#df_clean = df_clean.toDF(*[col.lower().replace(" ", "_") for col in df_clean.co

# Rename columns to snake_case
new_column_names = [c.lower().replace(" ", "_") for c in df_clean.columns]
df_clean = df_clean.toDF(*new_column_names)


#df_clean.show(20)


# Write to Parquet
#df_clean.write.mode("overwrite").parquet("Files/customer_data_cleaned_parquet")

# Register as table
#df_clean.write.mode("overwrite").saveAsTable("customer_data_cleaned")
```

StatementMeta(, 9f38f95a-4ff1-4a42-bcd1-c8f923dbf6bc, 5, Finished, Available, Fin
ished)

Data Cleaning and Rule Validation

- Invalid dates → parse and flag
-
- Numeric fields → cast and flag errors
-
- Date formatting → standardize to yyyy-MM-dd

In [5]:
```
# Clean and validate
valid_countries = ['United States', 'United Kingdom', 'Nigeria', 'India', 'Germa
valid_subscriptions = ["Free", "Premium", "Trial", "Enterprise", "Unknown"]

df_cleaned2 = df_clean \
    .withColumn("signup_date_parsed", to_date("signup_date", "yyyy-MM-dd")) \
    .withColumn("signup_date_valid", col("signup_date_parsed").isNotNull()) \
    .withColumn("age_clean", col("age").cast(IntegerType())) \
    .withColumn("age_valid", col("age_clean").isNotNull() & (col("age_clean") >
    .withColumn("income_clean", regexp_replace("income", "[$,]", "").cast(Double
    .withColumn("income_valid", col("income_clean").isNotNull() & (col("income_c
    .withColumn("email_valid", col("email").rlike("^[A-Za-z0-9._%+-]+@[A-Za-z0-9
    .withColumn("country_code_valid", col("country_code").isin(valid_countries))
    .withColumn("phone_valid", col("phone_number").rlike("^[+()0-9xX.-]{7,}$"))
```

```
    .withColumn("subscription_valid", col("subscription_type").isin(valid_subscr
    .withColumn("last_login_parsed", to_timestamp("last_login"))    \
    .withColumn("last_login_valid", col("last_login_parsed").isNotNull()) \
    .withColumn("referral_code_valid", col("referral_code").rlike("^REF[A-Za-z0-

#df_cleaned2.show(10)
```

StatementMeta(, 9f38f95a-4ff1-4a42-bcd1-c8f923dbf6bc, 6, Finished, Available, Fin
ished)

Step 3: Error Logging

Capture rows with issues and reasons:

data validation pattern:

Filter invalid data

Label each invalid record with the reason

Save or review these errors separately for debugging and reporting

In [6]:
```
# Flag invalid rows
df_errors = df_cleaned2.filter(~(col("signup_date_valid") & col("age_valid") & c
    .withColumn("error_reason", when(~col("signup_date_valid"), lit("Invalid dat
                .when(~col("age_valid"), lit("Invalid age"))
                .when(~col("income_valid"), lit("Invalid income"))
                .when(~col("phone_valid"), lit("Invalid phone number")))

# Save errors to a separate folder
df_errors.write.mode("overwrite").parquet("Files/customer_data_errors")
#df_errors.show(20)
```

StatementMeta(, 9f38f95a-4ff1-4a42-bcd1-c8f923dbf6bc, 7, Finished, Available, Fin
ished)

Step 4: Filter and Save Clean Data

- Keep rows only if all three columns are True:
-
- signup_date_valid is True (date is valid)
-
- age_valid is True (age is valid)
-
- income_valid is True (income is valid)
-
- Exclude rows with any invalid values.

In [7]:
```
df_valid = df_cleaned2.filter(col("signup_date_valid") & col("age_valid") & col(
    .select("customer_id", "email", "signup_date_parsed", "age_clean", "income_c

df_valid.write.mode("overwrite").parquet("Files/customer_data_cleaned")

#df_valid.show(truncate=False)
```

StatementMeta(, 9f38f95a-4ff1-4a42-bcd1-c8f923dbf6bc, 8, Finished, Available, Fin
ished)

🧠 Step 5: Register as a Table - Save into Delta Table

In [8]:
```
'''
# In Spark with Delta Lake (which Fabric uses), DROP TABLE removes the table ref
# but not necessarily the underlying data files in the Lakehouse. This is why I

spark.sql("DROP TABLE IF EXISTS customer_data_cleaned")   #issue with the table
df_valid.write.mode("overwrite").saveAsTable("customer_data_cleaned")

#df_valid.write.mode("overwrite").saveAsTable("customer_data_cleaned")
df_valid.show(truncate=False)
'''
#To really re-create the schema with the new schema in the dataframe in case the
spark.conf.set("spark.databricks.delta.schema.autoMerge.enabled", "true")

df_valid.write \
    .option("overwriteSchema", "true") \
    .mode("overwrite") \
    .saveAsTable("customer_data_cleaned")

#df_valid.show(10)
```

StatementMeta(, 9f38f95a-4ff1-4a42-bcd1-c8f923dbf6bc, 9, Finished, Available, Finished)

Calculated Column for our analytics in reporting for bin, groupings

In [9]:
```
df = spark.read.table("customer_data_cleaned")

df = df \
.withColumn("income_bin", when(df["income_clean"] < 20000, "<20k")
     .when(df["income_clean"] <= 50000, "20k-50k")
     .when(df["income_clean"] <= 100000, "50k-100k")
     .otherwise(">100k")
) \
.withColumn("Age_group", when(df["age_clean"] < 18, "Under 18")
     .when(df["age_clean"] <= 30, "18-30")
     .when(df["age_clean"] <= 40, "31-40")
     .when(df["age_clean"] <= 50, "41-50")
     .when(df["age_clean"] <= 60, "51-60")
     .when(df["age_clean"] <= 70, "61-70")
     .otherwise("70+")
) \
.withColumn("Age_group_sort", when(df["age_clean"] < 18, 1)
     .when(df["age_clean"] <= 30, 2)
     .when(df["age_clean"] <= 40, 3)
     .when(df["age_clean"] <= 50, 4)
     .when(df["age_clean"] <= 60, 5)
     .when(df["age_clean"] <= 70, 6)
     .otherwise(7)
)
'''
df = df.withColumn("Age_group", when(df["age_clean"] < 18, "Under 18")
     .when(df["age_clean"] <= 30, "18-30")
     .when(df["age_clean"] <= 40, "31-40")
     .when(df["age_clean"] <= 50, "41-50")
     .when(df["age_clean"] <= 60, "51-60")
     .when(df["age_clean"] <= 70, "61-70")
     .otherwise("70+")
```

```
)
'''
df.write \
    .option("overwriteSchema", "true") \
    .mode("overwrite") \
    .saveAsTable("customer_data_cleaned")
```

StatementMeta(, 9f38f95a-4ff1-4a42-bcd1-c8f923dbf6bc, 10, Finished, Available, Finished)

Create calculated columns for dates in our date dimension

In [10]:
```
df = spark.read.table("customer_data_cleaned")

df = df.withColumn("Year", year(df["signup_date_parsed"])) \
       .withColumn("Month", date_format(df["signup_date_parsed"], "MMMM")) \
       .withColumn("Quarter", concat(lit("Q"), quarter(df["signup_date_parsed"])
       .withColumn("Day", dayofmonth(df["signup_date_parsed"]))

df = df.withColumn("FiscalYear",
    when(month("signup_date_parsed") >= 4, year("signup_date_parsed")).otherwise
)

df = df.withColumn("FiscalMonth",
    when(month("signup_date_parsed") >= 4, month("signup_date_parsed") - 3).othe
)

df = df.withColumn("FiscalQuarter",
    when(month("signup_date_parsed").between(4, 6), "Q1")
    .when(month("signup_date_parsed").between(7, 9), "Q2")
    .when(month("signup_date_parsed").between(10, 12), "Q3")
    .otherwise("Q4")
)
```

StatementMeta(, 9f38f95a-4ff1-4a42-bcd1-c8f923dbf6bc, 11, Finished, Available, Finished)

Fetch data from api for UK Bank holiday

In [11]:
```
import requests
from datetime import datetime

response = requests.get("https://www.gov.uk/bank-holidays.json")
holidays = response.json()["england-and-wales"]["events"]

uk_holiday_dates = [datetime.strptime(h["date"], "%Y-%m-%d").date() for h in hol
```

StatementMeta(, 9f38f95a-4ff1-4a42-bcd1-c8f923dbf6bc, 12, Finished, Available, Finished)

Convert to spark DataFrame

In [12]:
```
holiday_df = spark.createDataFrame([(d,) for d in uk_holiday_dates], ["HolidayDa
```

StatementMeta(, 9f38f95a-4ff1-4a42-bcd1-c8f923dbf6bc, 13, Finished, Available, Finished)

Join Holiday date to Table date to add a flag to indicate holiday dates in UK

In [13]:
```
df = df.join(holiday_df, df["signup_date_parsed"] == holiday_df["HolidayDate"],
       .withColumn("IsUKHoliday", when(holiday_df["HolidayDate"].isNotNull(), Tr
```

StatementMeta(, 9f38f95a-4ff1-4a42-bcd1-c8f923dbf6bc, 14, Finished, Available, Finished)

Write our final result to table

In [14]:
```
df.write \
    .option("overwriteSchema", "true") \
    .mode("overwrite") \
    .saveAsTable("customer_data_cleaned")
```

StatementMeta(, 9f38f95a-4ff1-4a42-bcd1-c8f923dbf6bc, 15, Finished, Available, Finished)

In [15]:
```
df = spark.sql("SELECT * FROM laserengravelakehouse.customer_data_cleaned LIMIT
display(df)
```

StatementMeta(, 9f38f95a-4ff1-4a42-bcd1-c8f923dbf6bc, 16, Finished, Available, Finished)
SynapseWidget(Synapse.DataFrame, a2ef5e74-3bda-4aed-ab86-98a225e6ddc0)